

# Package ‘vtreat’

July 20, 2018

**Type** Package

**Title** A Statistically Sound 'data.frame' Processor/Conditioner

**Version** 1.3.0

**Date** 2018-07-20

**URL** <https://github.com/WinVector/vtreat/>,  
<https://winvector.github.io/vtreat/>

**BugReports** <https://github.com/WinVector/vtreat/issues>

**Maintainer** John Mount <jmount@win-vector.com>

**Description** A 'data.frame' processor/conditioner that prepares real-world data for predictive modeling in a statistically sound manner. 'vtreat' prepares variables so that data has fewer exceptional cases, making it easier to safely use models in production. Common problems 'vtreat' defends against: 'Inf', 'NA', too many categorical levels, rare categorical levels, and new categorical levels (levels seen during application, but not during training). Reference: ``vtreat': a data.frame Processor for Predictive Modeling", 'Zumel', 'Mount', 2016, DOI:10.5281/zenodo.1173314.

**License** GPL-3

**Depends** R (>= 3.2.1)

**Imports** stats, parallel, wrapr (>= 1.5.0)

**Suggests** rquery (>= 0.5.0), rqdatatable, testthat, knitr, rmarkdown, data.table, ggplot2, DBI, RSQLite, datasets

**LazyData** true

**VignetteBuilder** knitr

**RoxygenNote** 6.0.1

**ByteCompile** true

**NeedsCompilation** no

**Author** John Mount [aut, cre],  
Nina Zumel [aut],  
Win-Vector LLC [cph]

**Repository** CRAN

**Date/Publication** 2018-07-20 15:20:02 UTC

**R topics documented:**

as_rquery_plan	2
buildEvalSets	4
designTreatmentsC	5
designTreatmentsN	7
designTreatmentsZ	9
design_missingness_treatment	11
flatten_fn_list	12
format.vtreatment	12
getSplitPlanAppLabels	13
kWayCross Validation	13
kWayStratifiedY	14
kWayStratifiedYReplace	15
makekWayCross ValidationGroupedByColumn	16
mkCrossFrameCExperiment	17
mkCrossFrameMExperiment	19
mkCrossFrameNExperiment	20
novel_value_summary	22
oneWayHoldout	23
prepare	24
prepare.multinomial_plan	24
prepare.simple_plan	25
prepare.treatmentplan	26
print.multinomial_plan	28
print.simple_plan	28
print.treatmentplan	29
print.vtreatment	29
problemAppPlan	30
rqdatatable_prepare	30
rquery_prepare	31
track_values	32
vnames	33
vorig	34
vtreat	34
<b>Index</b>	<b>35</b>

---

as_rquery_plan	<i>Convert vtreatment plans into a sequence of rquery operations.</i>
----------------	---

---

**Description**

Convert vtreatment plans into a sequence of rquery operations.

**Usage**

```
as_rquery_plan(treatmentplans, ..., var_restriction = NULL)
```

**Arguments**

treatmentplans vtreat treatment plan or list of vtreat treatment plan sharing same outcome and outcome type.

... not used, force any later arguments to bind to names.

var\_restriction character, if not null restrict to producing these variables.

**Value**

list(optree\_generator (ordered list of functions), temp\_tables (named list of tables))

**See Also**

[rquery\\_prepare](#)

**Examples**

```
if(requireNamespace("rquery", quietly = TRUE)) {
  dTrainC <- data.frame(x= c('a', 'a', 'a', 'b', NA, 'b'),
                      z= c(1, 2, NA, 4, 5, 6),
                      y= c(FALSE, FALSE, TRUE, FALSE, TRUE, TRUE),
                      stringsAsFactors = FALSE)
  dTrainC$id <- seq_len(nrow(dTrainC))
  treatmentsC <- designTreatmentsC(dTrainC, c("x", "z"), 'y', TRUE)
  print(prepare(treatmentsC, dTrainC))
  rqplan <- as_rquery_plan(list(treatmentsC))
  ops <- flatten_fn_list(rquery::local_td(dTrainC), rqplan$optree_generators)
  cat(format(ops))
  if(requireNamespace("rqdatatable", quietly = TRUE)) {
    treated <- rqdatatable::ex_data_table(ops, tables = rqplan$tables)
    print(treated[])
  }
  if(requireNamespace("DBI", quietly = TRUE) &&
      requireNamespace("RSQLite", quietly = TRUE)) {
    db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
    source_data <- rquery::rq_copy_to(db, "dTrainC", dTrainC,
                                     overwrite = TRUE, temporary = TRUE)

    rest <- rquery_prepare(db, rqplan, source_data, "dTreatedC",
                          extracols = "id",
                          print_sql = FALSE)
    resd <- DBI::dbReadTable(db, rest$table_name)
    print(resd)

    rquery::rq_remove_table(db, source_data$table_name)
    rquery::rq_remove_table(db, rest$table_name)
    DBI::dbDisconnect(db)
  }
}
```

---

buildEvalSets	<i>Build set carve-up for out-of sample evaluation.</i>
---------------	---

---

### Description

Return a carve-up of `seq_len(nRows)`. Very useful for any sort of nested model situation (such as data prep, stacking, or super-learning).

### Usage

```
buildEvalSets(nRows, ..., dframe = NULL, y = NULL, splitFunction = NULL,
              nSplits = 3)
```

### Arguments

<code>nRows</code>	scalar, $\geq 1$ number of rows to sample from.
<code>...</code>	no additional arguments, declared to forced named binding of later arguments.
<code>dframe</code>	(optional) original data.frame, passed to user <code>splitFunction</code> .
<code>y</code>	(optional) numeric vector, outcome variable (possibly to stratify on), passed to user <code>splitFunction</code> .
<code>splitFunction</code>	(optional) function taking arguments <code>nSplits</code> , <code>nRows</code> , <code>dframe</code> , and <code>y</code> ; returning a user desired split.
<code>nSplits</code>	integer, target number of splits.

### Details

Also sets attribute "splitmethod" on return value that describes how the split was performed. `attr(returnValue,'splitmethod')` is one of: 'notsplit' (data was not split; corner cases like single row data sets), 'oneway' (leave one out holdout), 'kwaycross' (a simple partition), 'userfunction' (user supplied function was actually used), or a user specified attribute. Any user desired properties (such as stratification on `y`, or preservation of groups designated by original data row numbers) may not apply unless you see that 'userfunction' has been used.

The intent is the user `splitFunction` only needs to handle "easy cases" and maintain user invariants. If the user `splitFunction` returns `NULL`, throws, or returns an unacceptable carve-up then `vtreat::buildEvalSets` returns its own eval set plan. The signature of `splitFunction` should be `splitFunction(nRows,nSplits,dframe,y)` where `nSplits` is the number of pieces we want in the carve-up, `nRows` is the number of rows to split, `dframe` is the original dataframe (useful for any group control variables), and `y` is a numeric vector representing outcome (useful for outcome stratification).

Note that `buildEvalSets` may not always return a partition (such as one row dataframes), or if the user split function chooses to make rows eligible for application a different number of times.

### Value

list of lists where the app portion of the sub-lists is a disjoint carve-up of `seq_len(nRows)` and each list as a train portion disjoint from app.

**See Also**

[kWayCrossValidation](#), [kWayStratifiedY](#), and [makekWayCrossValidationGroupedByColumn](#)

**Examples**

```
# use
buildEvalSets(200)

# longer example
# helper fns
# fit models using experiment plan to estimate out of sample behavior
fitModelAndApply <- function(trainData,applicationData) {
  model <- lm(y~x,data=trainData)
  predict(model,newdata=applicationData)
}
simulateOutOfSampleTrainEval <- function(d,fitApplyFn) {
  eSets <- buildEvalSets(nrow(d))
  evals <- lapply(eSets,
    function(ei) { fitApplyFn(d[ei$train,],d[ei$app,]) })
  pred <- numeric(nrow(d))
  for(eii in seq_len(length(eSets))) {
    pred[eSets[[eii]]$app] <- evals[[eii]]
  }
  pred
}

# run the experiment
set.seed(2352356)
# example data
d <- data.frame(x=rnorm(5),y=rnorm(5),
  outOfSampleEst=NA,inSampleEst=NA)

# fit model on all data
d$inSampleEst <- fitModelAndApply(d,d)
# compute in-sample R^2 (above zero, falsely shows a
# relation until we adjust for degrees of freedom)
1-sum((d$y-d$inSampleEst)^2)/sum((d$y-mean(d$y))^2)

d$outOfSampleEst <- simulateOutOfSampleTrainEval(d,fitModelAndApply)
# compute out-sample R^2 (not positive,
# evidence of no relation)
1-sum((d$y-d$outOfSampleEst)^2)/sum((d$y-mean(d$y))^2)
```

## Description

Function to design variable treatments for binary prediction of a categorical outcome. Data frame is assumed to have only atomic columns except for dates (which are converted to numeric). Note: re-encoding high cardinality categorical variables can introduce undesirable nested model bias, for such data consider using [mkCrossFrameCExperiment](#).

## Usage

```
designTreatmentsC(dframe, varlist, outcomename, outcometarget, ...,
  weights = c(), minFraction = 0.02, smFactor = 0, rareCount = 0,
  rareSig = NULL, collarProb = 0, codeRestriction = NULL,
  customCoders = NULL, splitFunction = NULL, ncross = 3,
  forceSplit = FALSE, catScaling = FALSE, verbose = TRUE,
  parallelCluster = NULL, use_parallel = TRUE)
```

## Arguments

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
outcomename	Name of column holding outcome variable. <code>dframe[[outcomename]]</code> must be only finite non-missing values.
outcometarget	Value/level of outcome to be considered "success", and there must be a cut such that <code>dframe[[outcomename]]==outcometarget</code> at least twice and <code>dframe[[outcomename]]!=outcometarget</code> at least twice.
...	no additional arguments, declared to forced named binding of later arguments
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if <code>doCollar</code> is set during <a href="#">prepare.treatmentplan</a> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md</a> ).
splitFunction	(optional) see <code>vtreat::buildEvalSets</code> .
ncross	optional scalar $\geq 2$ number of cross validation splits use in rescoring complex variables.

forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
catScaling	optional, if TRUE use glm() linkspace, if FALSE use lm() for scaling.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods (when parallel cluster is set).

### Details

The main fields are mostly vectors with names (all with the same names in the same order):

- vars : (character array without names) names of variables (in same order as names on the other diagnostic vectors) - varMoves : logical TRUE if the variable varied during hold out scoring, only variables that move will be in the treated frame - #' - sig : an estimate significance of effect

See the vtreat vignette for a bit more detail and a worked example.

### Value

treatment plan (for use with prepare)

### See Also

[prepare.treatmentplan](#), [designTreatmentsN](#), [designTreatmentsZ](#), [mkCrossFrameCEExperiment](#)

### Examples

```
dTrainC <- data.frame(x=c('a','a','a','b','b','b'),
  z=c(1,2,3,4,5,6),
  y=c(FALSE,FALSE,TRUE,FALSE,TRUE,TRUE))
dTestC <- data.frame(x=c('a','b','c',NA),
  z=c(10,20,30,NA))
treatmentsC <- designTreatmentsC(dTrainC,colnames(dTrainC),'y',TRUE)
dTrainCTreated <- prepare(treatmentsC,dTrainC,pruneSig=0.99)
dTestCTreated <- prepare(treatmentsC,dTestC,pruneSig=0.99)
```

---

designTreatmentsN      *build all treatments for a data frame to predict a numeric outcome*

---

### Description

Function to design variable treatments for binary prediction of a numeric outcome. Data frame is assumed to have only atomic columns except for dates (which are converted to numeric). Note: each column is processed independently of all others. Note: re-encoding high cardinality categorical variables can introduce undesirable nested model bias, for such data consider using [mkCrossFrameNExperiment](#).

**Usage**

```
designTreatmentsN(dframe, varlist, outcomename, ..., weights = c(),
  minFraction = 0.02, smFactor = 0, rareCount = 0, rareSig = NULL,
  collarProb = 0, codeRestriction = NULL, customCoders = NULL,
  splitFunction = NULL, ncross = 3, forceSplit = FALSE, verbose = TRUE,
  parallelCluster = NULL, use_parallel = TRUE)
```

**Arguments**

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
outcomename	Name of column holding outcome variable. <code>dframe[[outcomename]]</code> must be only finite non-missing values and there must be a cut such that <code>dframe[[outcomename]]</code> is both above the cut at least twice and below the cut at least twice.
...	no additional arguments, declared to forced named binding of later arguments
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if <code>doCollar</code> is set during <code>prepare.treatmentplan</code> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md</a> ).
splitFunction	(optional) see <code>vtreat::buildEvalSets</code> .
ncross	optional scalar $\geq 2$ number of cross validation splits use in rescoring complex variables.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package <code>parallel</code> or package <code>snow</code> .
use_parallel	logical, if TRUE use parallel methods (when parallel cluster is set).



**Details**

The main fields are mostly vectors with names (all with the same names in the same order):

- vars : (character array without names) names of variables (in same order as names on the other diagnostic vectors) - varMoves : logical TRUE if the variable varied during hold out scoring, only variables that move will be in the treated frame - sig : an estimate significance of effect

See the vtreat vignette for a bit more detail and a worked example.

**Value**

treatment plan (for use with prepare)

**See Also**

[prepare.treatmentplan](#), [designTreatmentsC](#), [designTreatmentsZ](#), [mkCrossFrameNExperiment](#)

**Examples**

```
dTrainN <- data.frame(x=c('a','a','a','a','b','b','b'),
  z=c(1,2,3,4,5,6,7),y=c(0,0,0,1,0,1,1))
dTestN <- data.frame(x=c('a','b','c',NA),
  z=c(10,20,30,NA))
treatmentsN = designTreatmentsN(dTrainN,colnames(dTrainN),'y')
dTrainNTreated <- prepare(treatmentsN,dTrainN,pruneSig=0.99)
dTestNTreated <- prepare(treatmentsN,dTestN,pruneSig=0.99)
```

---

designTreatmentsZ      *Design variable treatments with no outcome variable.*

---

**Description**

Data frame is assumed to have only atomic columns except for dates (which are converted to numeric). Note: each column is processed independently of all others.

**Usage**

```
designTreatmentsZ(dframe, varlist, ..., minFraction = 0, weights = c(),
  rareCount = 0, collarProb = 0, codeRestriction = NULL,
  customCoders = NULL, verbose = TRUE, parallelCluster = NULL,
  use_parallel = TRUE)
```

**Arguments**

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
...	no additional arguments, declared to forced named binding of later arguments
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
weights	optional training weights for each row
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <code>prepare.treatmentplan</code> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md</a> ).
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods (if parallel cluster is set).

**Details**

The main fields are mostly vectors with names (all with the same names in the same order):

- vars : (character array without names) names of variables (in same order as names on the other diagnostic vectors) - varMoves : logical TRUE if the variable varied during hold out scoring, only variables that move will be in the treated frame

See the vtreat vignette for a bit more detail and a worked example.

**Value**

treatment plan (for use with prepare)

**See Also**

[prepare.treatmentplan](#), [designTreatmentsC](#), [designTreatmentsN](#)

**Examples**

```
dTrainZ <- data.frame(x=c('a', 'a', 'a', 'a', 'b', 'b', NA, 'e', 'e'),
  z=c(1, 2, 3, 4, 5, 6, 7, NA, 9))
dTestZ <- data.frame(x=c('a', 'x', 'c', NA),
  z=c(10, 20, 30, NA))
```

```
treatmentsZ = designTreatmentsZ(dTrainZ, colnames(dTrainZ),
  rareCount=0)
dTrainZTreated <- prepare(treatmentsZ, dTrainZ)
dTestZTreated <- prepare(treatmentsZ, dTestZ)
```

---

design\_missingness\_treatment

*Design a simple treatment plan to indicate missingness and perform simple imputation.*

---

### Description

Design a simple treatment plan to indicate missingness and perform simple imputation.

### Usage

```
design_missingness_treatment(dframe, ..., varlist = colnames(dframe),
  invalid_mark = "_invalid_")
```

### Arguments

dframe	data.frame to drive design.
...	not used, forces later arguments to bind by name.
varlist	character, names of columns to process.
invalid_mark	character, name to use for NA levels and novel levels.

### Value

simple treatment plan.

### See Also

[prepare.simple\\_plan](#)

### Examples

```
d <- wrapr::build_frame(
  "x1", "x2", "x3" |
  1 , 4 , "A" |
  NA , 5 , "B" |
  3 , 6 , NA )

plan <- design_missingness_treatment(d)
prepare(plan, d)

prepare(plan, data.frame(x1=NA, x2=NA, x3="E"))
```

---

flatten_fn_list	<i>Flatten a list of functions onto d.</i>
-----------------	--

---

**Description**

Flatten a list of functions onto d.

**Usage**

```
flatten_fn_list(d, fnlist)
```

**Arguments**

d	object (usually a data source)
fnlist	a list of functions

**Value**

fnlist[[length(fnlist)]](flatten\_fn\_list(d, fnlist[[-length(fnlist)]]) (or d if length(fnlist)<1)

**See Also**

[as\\_rquery\\_plan](#)

---

format.vtreatment	<i>Display treatment plan.</i>
-------------------	--------------------------------

---

**Description**

Display treatment plan.

**Usage**

```
## S3 method for class 'vtreatment'
format(x, ...)
```

**Arguments**

x	treatment plan
...	additional args (to match general signature).

---

getSplitPlanAppLabels *read application labels off a split plan.*

---

**Description**

read application labels off a split plan.

**Usage**

```
getSplitPlanAppLabels(nRow, plan)
```

**Arguments**

nRow	number of rows in original data.frame.
plan	split plan

**Value**

vector of labels

**See Also**

[kWayCrossValidation](#), [kWayStratifiedY](#), and [makekWayCrossValidationGroupedByColumn](#)

**Examples**

```
plan <- kWayStratifiedY(3,2,NULL,NULL)
getSplitPlanAppLabels(3,plan)
```

---

[kWayCrossValidation](#) *k-fold cross validation, a splitFunction in the sense of vtreat::buildEvalSets*

---

**Description**

k-fold cross validation, a splitFunction in the sense of vtreat::buildEvalSets

**Usage**

```
kWayCrossValidation(nRows, nSplits, dframe, y)
```

**Arguments**

nRows	number of rows to split (>1).
nSplits	number of groups to split into (>1,<=nRows).
dframe	original data frame (ignored).
y	numeric outcome variable (ignored).

**Value**

split plan

**Examples**

```
kWayCrossValidation(7,2,NULL,NULL)
```

---

kWayStratifiedY	<i>k-fold cross validation stratified on y, a splitFunction in the sense of vtreat::buildEvalSets</i>
-----------------	---

---

**Description**

k-fold cross validation stratified on y, a splitFunction in the sense of vtreat::buildEvalSets

**Usage**

```
kWayStratifiedY(nRows, nSplits, dframe, y)
```

**Arguments**

nRows	number of rows to split (>1)
nSplits	number of groups to split into (<nRows,>1).
dframe	original data frame (ignored).
y	numeric outcome variable try to have equidistributed in each split.

**Value**

split plan

**Examples**

```

set.seed(23255)
d <- data.frame(y=sin(1:100))
pStrat <- kWayStratifiedY(nrow(d),5,d,d$y)
problemAppPlan(nrow(d),5,pStrat,TRUE)
d$stratGroup <- vtreat::getSplitPlanAppLabels(nrow(d),pStrat)
pSimple <- kWayCrossValidation(nrow(d),5,d,d$y)
problemAppPlan(nrow(d),5,pSimple,TRUE)
d$simpleGroup <- vtreat::getSplitPlanAppLabels(nrow(d),pSimple)
summary(tapply(d$y,d$simpleGroup,mean))
# ggplot(data=d,aes(x=y,color=as.factor(simpleGroup))) +
#   geom_density() + ggtitle('simple grouping')
summary(tapply(d$y,d$stratGroup,mean))
# ggplot(data=d,aes(x=y,color=as.factor(stratGroup))) +
#   geom_density() + ggtitle('y-stratified grouping')

# # And you can (and should) use your own functions or libraries.
# splitFn <- function(nRows,nSplits,dframe,y) {
#   fullSeq <- seq_len(nRows)
#   part <- caret::createFolds(y=y,k=nSplits)
#   lapply(part,
#     function(appi) {
#       list(train=setdiff(fullSeq,appi),app=appi)
#     })
# }
# pCaret <- splitFn(nrow(d),5,d,d$y)
# problemAppPlan(nrow(d),5,pCaret,TRUE)
# d$caretGroup <- vtreat::getSplitPlanAppLabels(nrow(d),pCaret)
# ggplot(data=d,aes(x=y,color=as.factor(caretGroup))) +
#   geom_density() + ggtitle('caret::createFolds grouping')

```

---

kWayStratifiedYReplace

*k-fold cross validation stratified with replacement on y, a splitFunction in the sense of vtreat::buildEvalSets .*

---

**Description**

Build a k-fold cross validation sample where training sets are the same size as the original data, and built by sampling disjoint from test/application sets (sampled with replacement).

**Usage**

```
kWayStratifiedYReplace(nRows, nSplits, dframe, y)
```

**Arguments**

nRows	number of rows to split (>1)
nSplits	number of groups to split into (<nRows,>1).
dframe	original data frame (ignored).
y	numeric outcome variable try to have equidistributed in each split.

**Value**

split plan

**Examples**

```
set.seed(23255)
d <- data.frame(y=sin(1:100))
pStrat <- kWayStratifiedYReplace(nrow(d),5,d,d$y)
```

---

makekWayCrossValidationGroupedByColumn

*Build a k-fold cross validation splitter, respecting (never splitting) groupingColumn.*

---

**Description**

Build a k-fold cross validation splitter, respecting (never splitting) groupingColumn.

**Usage**

```
makekWayCrossValidationGroupedByColumn(groupingColumnName)
```

**Arguments**

groupingColumnName  
name of column to group by.

**Value**

splitting function in the sense of vtreat::buildEvalSets.



## Examples

```
d <- data.frame(y=sin(1:100))
d$group <- floor(seq_len(nrow(d))/5)
splitter <- makekWayCrossValidationGroupedByColumn('group')
split <- splitter(nrow(d),5,d,d$y)
d$splitLabel <- vtreat::getSplitPlanAppLabels(nrow(d),split)
rowSums(table(d$group,d$splitLabel)>0)
```

---

```
mkCrossFrameCExperiment
```

*Run categorical cross-frame experiment.*

---

## Description

Builds a [designTreatmentsC](#) treatment plan and a data frame prepared from `dframe` that is "cross" in the sense each row is treated using a treatment plan built from a subset of `dframe` disjoint from the given row. The goal is to try to and supply a method of breaking nested model bias other than splitting into calibration, training, test sets.

## Usage

```
mkCrossFrameCExperiment(dframe, varlist, outcomename, outcometarget, ...,
  weights = c(), minFraction = 0.02, smFactor = 0, rareCount = 0,
  rareSig = 1, collarProb = 0, codeRestriction = NULL,
  customCoders = NULL, scale = FALSE, doCollar = FALSE,
  splitFunction = NULL, ncross = 3, forceSplit = FALSE,
  catScaling = FALSE, verbose = TRUE, parallelCluster = NULL,
  use_parallel = TRUE)
```

## Arguments

<code>dframe</code>	Data frame to learn treatments from (training data), must have at least 1 row.
<code>varlist</code>	Names of columns to treat (effective variables).
<code>outcomename</code>	Name of column holding outcome variable. <code>dframe[[outcomename]]</code> must be only finite non-missing values.
<code>outcometarget</code>	Value/level of outcome to be considered "success", and there must be a cut such that <code>dframe[[outcomename]]==outcometarget</code> at least twice and <code>dframe[[outcomename]]!=outcometarget</code> at least twice.
<code>...</code>	no additional arguments, declared to forced named binding of later arguments
<code>weights</code>	optional training weights for each row
<code>minFraction</code>	optional minimum frequency a categorical level must have to be converted to an indicator column.
<code>smFactor</code>	optional smoothing factor for impact coding models.

rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <code>prepare.treatmentplan</code> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md</a> ).
scale	optional if TRUE replace numeric variables with regression ("move to outcome-scale").
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
splitFunction	(optional) see <code>vtreat::buildEvalSets</code> .
ncross	optional scalar $\geq 2$ number of cross-validation rounds to design.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
catScaling	optional, if TRUE use <code>glm()</code> linkspace, if FALSE use <code>lm()</code> for scaling.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package <code>parallel</code> or package <code>snow</code> .
use_parallel	logical, if TRUE use parallel methods.

**Value**

list with treatments and crossFrame

**See Also**

`designTreatmentsC`, `designTreatmentsN`, `prepare.treatmentplan`

**Examples**

```
set.seed(23525)
zip <- paste('z',1:100)
N <- 200
d <- data.frame(zip=sample(zip,N,replace=TRUE),
                zip2=sample(zip,20,replace=TRUE),
                y=runif(N))
del <- runif(length(zip))
names(del) <- zip
d$y <- d$y + del[d$zip2]
d$yc <- d$y>=mean(d$y)
```

```

cC <- mkCrossFrameCExperiment(d,c('zip','zip2'),'yc',TRUE,
  rareCount=2,rareSig=0.9)
cor(as.numeric(cC$crossFrame$yc),cC$crossFrame$zip_catB) # poor
cor(as.numeric(cC$crossFrame$yc),cC$crossFrame$zip2_catB) # better
treatments <- cC$treatments
dTrainV <- cC$crossFrame

```

---

mkCrossFrameMExperiment

*Function to build multi-outcome vtreat cross frame and treatment plan.*

---

## Description

Please see vignette("MultiClassVtreat", package = "vtreat") <https://winvector.github.io/vtreat/articles/MultiClassVtreat.html>.

## Usage

```

mkCrossFrameMExperiment(d, vars, y_name, ...,
  y_dependent_treatments = c("catB"), weights = c(), minFraction = 0.02,
  smFactor = 0, rareCount = 0, rareSig = 1, collarProb = 0,
  codeRestriction = NULL, customCoders = NULL, scale = FALSE,
  doCollar = FALSE, splitFunction = NULL, ncross = 3,
  forceSplit = FALSE, catScaling = FALSE, verbose = FALSE,
  parallelCluster = NULL, use_parallel = TRUE)

```

## Arguments

d	data to learn from
vars	character, vector of independent variable column names.
y_name	character, name of outcome column.
...	not used, declared to forced named binding of later arguments
y_dependent_treatments	character what treatment types to build per-outcome level.
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <a href="#">prepare.multinomial_plan</a> .

codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md</a> ).
scale	optional if TRUE replace numeric variables with regression ("move to outcome-scale").
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
splitFunction	(optional) see vtreat::buildEvalSets .
ncross	optional scalar $\geq 2$ number of cross-validation rounds to design.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
catScaling	optional, if TRUE use glm() linkspace, if FALSE use lm() for scaling.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods.

**Value**

list(cross\_frame, treatments\_0, treatments\_m)

**See Also**

[prepare.multinomial\\_plan](#)

---

mkCrossFrameNExperiment

*Run a numeric cross frame experiment.*

---

**Description**

Builds a [designTreatmentsN](#) treatment plan and a data frame prepared from dframe that is "cross" in the sense each row is treated using a treatment plan built from a subset of dframe disjoint from the given row. The goal is to try to and supply a method of breaking nested model bias other than splitting into calibration, training, test sets.

**Usage**

```
mkCrossFrameNExperiment(dframe, varlist, outcomename, ..., weights = c(),
  minFraction = 0.02, smFactor = 0, rareCount = 0, rareSig = 1,
  collarProb = 0, codeRestriction = NULL, customCoders = NULL,
  scale = FALSE, doCollar = FALSE, splitFunction = NULL, ncross = 3,
  forceSplit = FALSE, verbose = TRUE, parallelCluster = NULL,
  use_parallel = TRUE)
```

**Arguments**

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
outcomename	Name of column holding outcome variable. dframe[[outcomename]] must be only finite non-missing values and there must be a cut such that dframe[[outcomename]] is both above the cut at least twice and below the cut at least twice.
...	no additional arguments, declared to forced named binding of later arguments
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <code>prepare.treatmentplan</code> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/master/extras/CustomLevelCoders.md</a> ).
scale	optional if TRUE replace numeric variables with regression ("move to outcome-scale").
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
splitFunction	(optional) see <code>vtreat::buildEvalSets</code> .
ncross	optional scalar $\geq 2$ number of cross-validation rounds to design.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package <code>parallel</code> or package <code>snow</code> .
use_parallel	logical, if TRUE use parallel methods.

**Value**

treatment plan (for use with `prepare`)

**See Also**

[designTreatmentsC](#), [designTreatmentsN](#), [prepare.treatmentplan](#)

**Examples**

```

set.seed(23525)
zip <- paste('z',1:100)
N <- 200
d <- data.frame(zip=sample(zip,N,replace=TRUE),
                zip2=sample(zip,N,replace=TRUE),
                y=runif(N))
del <- runif(length(zip))
names(del) <- zip
d$y <- d$y + del[d$zip2]
d$yc <- d$y>=mean(d$y)
cN <- mkCrossFrameNExperiment(d,c('zip', 'zip2'),'y',
                             rareCount=2,rareSig=0.9)
cor(cN$crossFrame$y,cN$crossFrame$zip_catN) # poor
cor(cN$crossFrame$y,cN$crossFrame$zip2_catN) # better
treatments <- cN$treatments
dTrainV <- cN$crossFrame

```

---

novel\_value\_summary    *Report new/novel appearances of character values.*

---

**Description**

Report new/novel appearances of character values.

**Usage**

```
novel_value_summary(dframe, trackedValues)
```

**Arguments**

dframe	Data frame to inspect.
trackedValues	optional named list mapping variables to know values, allows warnings upon novel level appearances (see <a href="#">track_values</a> )

**Value**

frame of novel occurrences

**See Also**

[prepare.treatmentplan](#), [track\\_values](#)

**Examples**

```

set.seed(23525)
zip <- c(NA, paste('z', 1:10, sep = "_"))
N <- 10
d <- data.frame(zip = sample(zip, N, replace=TRUE),
                zip2 = sample(zip, N, replace=TRUE),
                y = runif(N))
dSample <- d[1:5, , drop = FALSE]
trackedValues <- track_values(dSample, c("zip", "zip2"))
novel_value_summary(d, trackedValues)

```

---

oneWayHoldout

*One way holdout, a splitFunction in the sense of vtreat::buildEvalSets.*


---

**Description**

Note one way holdout can leak target expected values, so it should not be preferred in nested modeling situations. Also, doesn't respect nSplits.

**Usage**

```
oneWayHoldout(nRows, nSplits, dframe, y)
```

**Arguments**

nRows	number of rows to split (integer >1).
nSplits	number of groups to split into (ignored).
dframe	original data frame (ignored).
y	numeric outcome variable (ignored).

**Value**

split plan

**Examples**

```
oneWayHoldout(3, NULL, NULL, NULL)
```

---

prepare	<i>Apply treatments and restrict to useful variables.</i>
---------	---

---

**Description**

Apply treatments and restrict to useful variables.

**Usage**

```
prepare(treatmentplan, dframe, ...)
```

**Arguments**

treatmentplan	Plan built by designTreatmentsC() or designTreatmentsN()
dframe	Data frame to be treated
...	no additional arguments, declared to forced named binding of later arguments

**See Also**

[prepare.treatmentplan](#), [prepare.simple\\_plan](#), [prepare.multinomial\\_plan](#)

---

```
prepare.multinomial_plan
```

*Function to apply mkCrossFrameMExperiment treatemnts.*

---

**Description**

Please see vignette("MultiClassVtreat", package = "vtreat") <https://winvector.github.io/vtreat/articles/MultiClassVtreat.html>.

**Usage**

```
## S3 method for class 'multinomial_plan'
prepare(treatmentplan, dframe, ...,
        pruneSig = NULL, scale = FALSE, doCollar = FALSE,
        varRestriction = NULL, codeRestriction = NULL, trackedValues = NULL,
        extracols = NULL, parallelCluster = NULL, use_parallel = TRUE)
```



**Arguments**

treatmentplan	multinomial_plan from mkCrossFrameMExperiment.
dframe	new data to process.
...	not used, declared to forced named binding of later arguments
pruneSig	suppress variables with significance above this level
scale	optional if TRUE replace numeric variables with single variable model regressions ("move to outcome-scale"). These have mean zero and (for variables with significant less than 1) slope 1 when regressed (lm for regression problems/glm for classification problems) against outcome.
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
varRestriction	optional list of treated variable names to restrict to
codeRestriction	optional list of treated variable codes to restrict to
trackedValues	optional named list mapping variables to know values, allows warnings upon novel level appearances (see <a href="#">track_values</a> )
extracols	extra columns to copy.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods.

**Value**

prepared data frame.

**See Also**

[mkCrossFrameMExperiment](#), [prepare](#)

---

prepare.simple\_plan    *Prepare a simple treatment.*

---

**Description**

Prepare a simple treatment.

**Usage**

```
## S3 method for class 'simple_plan'
prepare(treatmentplan, dframe, ...)
```

**Arguments**

treatmentplan A simple treatment plan.  
 dframe data.frame to be treated.  
 ... not used, present for S3 signature consistency.

**See Also**

[design\\_missingness\\_treatment](#), [prepare](#)

**Examples**

```
d <- wrapr::build_frame(
  "x1", "x2", "x3" |
  1 , 4 , "A" |
  NA , 5 , "B" |
  3 , 6 , NA )

plan <- design_missingness_treatment(d)
prepare(plan, d)

prepare(plan, data.frame(x1=NA, x2=NA, x3="E"))
```

---

prepare.treatmentplan *Apply treatments and restrict to useful variables.*

---

**Description**

Use a treatment plan to prepare a data frame for analysis. The resulting frame will have new effective variables that are numeric and free of NaN/NA. If the outcome column is present it will be copied over. The intent is that these frames are compatible with more machine learning techniques, and avoid a lot of corner cases (NA,NaN, novel levels, too many levels). Note: each column is processed independently of all others. Also copies over outcome if present. Note: treatmentplan's are not meant for long-term storage, a warning is issued if the version of vtreat that produced the plan differs from the version running prepare().

**Usage**

```
## S3 method for class 'treatmentplan'
prepare(treatmentplan, dframe, ..., pruneSig = NULL,
  scale = FALSE, doCollar = FALSE, varRestriction = NULL,
  codeRestriction = NULL, trackedValues = NULL, extracols = NULL,
  parallelCluster = NULL, use_parallel = TRUE)
```

**Arguments**

treatmentplan	Plan built by designTreatmentsC() or designTreatmentsN()
dframe	Data frame to be treated
...	no additional arguments, declared to forced named binding of later arguments
pruneSig	suppress variables with significance above this level
scale	optional if TRUE replace numeric variables with single variable model regressions ("move to outcome-scale"). These have mean zero and (for variables with significant less than 1) slope 1 when regressed (lm for regression problems/glm for classification problems) against outcome.
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
varRestriction	optional list of treated variable names to restrict to
codeRestriction	optional list of treated variable codes to restrict to
trackedValues	optional named list mapping variables to know values, allows warnings upon novel level appearances (see <a href="#">track_values</a> )
extracols	extra columns to copy.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods.

**Value**

treated data frame (all columns numeric- without NA, NaN)

**See Also**

[mkCrossFrameCExperiment](#), [mkCrossFrameNExperiment](#), [designTreatmentsC](#) [designTreatmentsN](#) [designTreatmentsZ](#), [prepare](#)

**Examples**

```
dTrainN <- data.frame(x= c('a','a','a','a','b','b','b'),
                    z= c(1,2,3,4,5,6,7),
                    y= c(0,0,0,1,0,1,1))
dTestN <- data.frame(x= c('a','b','c',NA),
                    z= c(10,20,30,NA))
treatmentsN = designTreatmentsN(dTrainN,colnames(dTrainN), 'y')
dTrainNTreated <- prepare(treatmentsN, dTrainN, pruneSig= 0.2)
dTestNTreated <- prepare(treatmentsN, dTestN, pruneSig= 0.2)

dTrainC <- data.frame(x= c('a','a','a','b','b','b'),
                    z= c(1,2,3,4,5,6),
                    y= c(FALSE,FALSE,TRUE,FALSE,TRUE,TRUE))
dTestC <- data.frame(x= c('a','b','c',NA),
                    z= c(10,20,30,NA))
```

```

treatmentsC <- designTreatmentsC(dTrainC, colnames(dTrainC), 'y', TRUE)
dTrainCTreated <- prepare(treatmentsC, dTrainC, varRestriction= c('z_clean'))
dTestCTreated <- prepare(treatmentsC, dTestC, varRestriction= c('z_clean'))

dTrainZ <- data.frame(x= c('a', 'a', 'a', 'b', 'b', 'b'),
                    z= c(1, 2, 3, 4, 5, 6))
dTestZ <- data.frame(x= c('a', 'b', 'c', NA),
                    z= c(10, 20, 30, NA))
treatmentsZ <- designTreatmentsZ(dTrainZ, colnames(dTrainZ))
dTrainZTreated <- prepare(treatmentsZ, dTrainZ, codeRestriction= c('lev'))
dTestZTreated <- prepare(treatmentsZ, dTestZ, codeRestriction= c('lev'))

```

---

```
print.multinomial_plan
```

*Print treatmentplan.*

---

### Description

Print treatmentplan.

### Usage

```
## S3 method for class 'multinomial_plan'
print(x, ...)
```

### Arguments

x	treatmentplan
...	additional args (to match general signature).

---

```
print.simple_plan
```

*Print treatmentplan.*

---

### Description

Print treatmentplan.

### Usage

```
## S3 method for class 'simple_plan'
print(x, ...)
```

### Arguments

x	treatmentplan
...	additional args (to match general signature).

---

`print.treatmentplan`    *Print treatmentplan.*

---

**Description**

Print treatmentplan.

**Usage**

```
## S3 method for class 'treatmentplan'  
print(x, ...)
```

**Arguments**

x                    treatmentplan  
...                  additional args (to match general signature).

**See Also**

[designTreatmentsC](#), [designTreatmentsN](#), [designTreatmentsZ](#), [prepare.treatmentplan](#)

---

`print.vtreatment`    *Print treatmentplan.*

---

**Description**

Print treatmentplan.

**Usage**

```
## S3 method for class 'vtreatment'  
print(x, ...)
```

**Arguments**

x                    treatmentplan  
...                  additional args (to match general signature).

**See Also**

[designTreatmentsC](#), [designTreatmentsN](#), [designTreatmentsZ](#), [prepare.treatmentplan](#)

---

problemAppPlan	<i>check if appPlan is a good carve-up of 1:nRows into nSplits groups</i>
----------------	---

---

**Description**

check if appPlan is a good carve-up of 1:nRows into nSplits groups

**Usage**

```
problemAppPlan(nRows, nSplits, appPlan, strictCheck)
```

**Arguments**

nRows	number of rows to carve-up
nSplits	number of sets to carve-up into
appPlan	carve-up to critique
strictCheck	logical, if true expect application data to be a carve-up and training data to be a maximal partition and to match nSplits.

**Value**

problem with carve-up (null if good)

**See Also**

[kWayCrossValidation](#), [kWayStratifiedY](#), and [makekWayCrossValidationGroupedByColumn](#)

**Examples**

```
plan <- kWayStratifiedY(3,2,NULL,NULL)
problemAppPlan(3,3,plan,TRUE)
```

---

rqdatatable_prepare	<i>Apply a treatment plan using rqdatatable.</i>
---------------------	--

---

**Description**

Apply a treatment plan using rqdatatable.

**Usage**

```
rqdatatable_prepare(rqplan, data_source, ..., partition_column = NULL,
  parallelCluster = NULL, use_parallel = use_parallel, extracols = NULL,
  non_join_mapping = FALSE, print_rquery = FALSE, env = parent.frame())
```

**Arguments**

rqplan	an query plan produced by <code>as_rquery_plan()</code> .
data_source	a <code>data.frame</code> .
...	force later arguments to bind by name.
partition_column	character name of column to partition work by.
parallelCluster	a cluster object, created by package <code>parallel</code> or by package <code>snow</code> . If <code>NULL</code> , use the registered default cluster.
use_parallel	logical, if <code>TRUE</code> use parallel cluster (when available).
extracols	extra columns to copy.
non_join_mapping	logical, if <code>TRUE</code> use non-join based column mapping.
print_rquery	logical, if <code>TRUE</code> print the rquery ops.
env	environment to work in.

**Value**

treated data.

**See Also**

[as\\_rquery\\_plan](#), [rquery\\_prepare](#)

---

rquery_prepare	<i>Materialize a treated data frame remotely.</i>
----------------	---

---

**Description**

Materialize a treated data frame remotely.

**Usage**

```
rquery_prepare(db, rqplan, data_source, result_table_name, ...,
  extracols = NULL, temporary = FALSE, overwrite = TRUE,
  print_rquery = FALSE, print_sql = FALSE)
```

```
materialize_treated(db, rqplan, data_source, result_table_name, ...,
  extracols = NULL, temporary = FALSE, overwrite = TRUE,
  print_rquery = FALSE, print_sql = FALSE)
```

**Arguments**

db	a db handle.
rqlan	an query plan produced by <code>as_rquery_plan()</code> .
data_source	relop, data source (usually a <code>relop_table_source</code> ).
result_table_name	character, table name to land result in
...	force later arguments to bind by name.
extracols	extra columns to copy.
temporary	logical, if TRUE try to make result temporary.
overwrite	logical, if TRUE try to overwrite result.
print_rquery	logical, if TRUE print the rquery ops.
print_sql	logical, if TRUE print the SQL.

**Value**

description of treated table.

**Functions**

- `materialize_treated`: old name for `rquery_prepare` function

**See Also**

[as\\_rquery\\_plan](#), [rqdatatable\\_prepare](#)

---

track_values	<i>Track unique character values for variables.</i>
--------------	---

---

**Description**

Builds lists of observed unique character values of varlist variables from the data frame.

**Usage**

```
track_values(dframe, varlist)
```

**Arguments**

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).

**Value**

named list of values seen.



**See Also**

[prepare.treatmentplan](#), [novel\\_value\\_summary](#)

**Examples**

```
set.seed(23525)
zip <- c(NA, paste('z', 1:100, sep = "_"))
N <- 500
d <- data.frame(zip = sample(zip, N, replace=TRUE),
                zip2 = sample(zip, N, replace=TRUE),
                y = runif(N))
dSample <- d[1:300, , drop = FALSE]
tplan <- designTreatmentsN(dSample,
                          c("zip", "zip2"), "y",
                          verbose = FALSE)
trackedValues <- track_values(dSample, c("zip", "zip2"))
# don't normally want to catch warnings,
# doing it here as this is an example
# and must not have unhandled warnings.
tryCatch(
  prepare(tplan, d, trackedValues = trackedValues),
  warning = function(w) { cat(paste(w, collapse = "\n")) })
```

---

vnames

*New treated variable names from a treatmentplan\$treatment item.*


---

**Description**

New treated variable names from a treatmentplan\$treatment item.

**Usage**

```
vnames(x)
```

**Arguments**

x                    vtreatment item

**See Also**

[designTreatmentsC](#) [designTreatmentsN](#) [designTreatmentsZ](#)

---

vorig	<i>Original variable name from a treatmentplan\$treatment item.</i>
-------	---

---

**Description**

Original variable name from a treatmentplan\$treatment item.

**Usage**

```
vorig(x)
```

**Arguments**

x vtreatment item.

**See Also**

[designTreatmentsC](#) [designTreatmentsN](#) [designTreatmentsZ](#)

---

vtreat	<i>vtreat: A Statistically Sound 'data.frame' Processor/Conditioner</i>
--------	---

---

**Description**

A 'data.frame' processor/conditioner that prepares real-world data for predictive modeling in a statistically sound manner. 'vtreat' prepares variables so that data has fewer exceptional cases, making it easier to safely use models in production. Common problems 'vtreat' defends against: 'Inf', 'NA', too many categorical levels, rare categorical levels, and new categorical levels (levels seen during application, but not during training). 'vtreat::prepare' should be used as you would use 'model.matrix'.

**Details**

For more information:

- `vignette('vtreat', package='vtreat')`
- `vignette(package='vtreat')`
- Website: <https://github.com/WinVector/vtreat>

# Index

as\_rquery\_plan, [2](#), [12](#), [31](#), [32](#)

buildEvalSets, [4](#)

design\_missingness\_treatment, [11](#), [26](#)

designTreatmentsC, [5](#), [9](#), [10](#), [17](#), [18](#), [21](#), [27](#),  
[29](#), [33](#), [34](#)

designTreatmentsN, [7](#), [7](#), [10](#), [18](#), [20](#), [21](#), [27](#),  
[29](#), [33](#), [34](#)

designTreatmentsZ, [7](#), [9](#), [9](#), [27](#), [29](#), [33](#), [34](#)

flatten\_fn\_list, [12](#)

format.vtreatment, [12](#)

getSplitPlanAppLabels, [13](#)

kWayCrossValidation, [5](#), [13](#), [13](#), [30](#)

kWayStratifiedY, [5](#), [13](#), [14](#), [30](#)

kWayStratifiedYReplace, [15](#)

makekWayCrossValidationGroupedByColumn,  
[5](#), [13](#), [16](#), [30](#)

materialize\_treated (rquery\_prepare), [31](#)

mkCrossFrameCExperiment, [6](#), [7](#), [17](#), [27](#)

mkCrossFrameMExperiment, [19](#), [25](#)

mkCrossFrameNExperiment, [7](#), [9](#), [20](#), [27](#)

novel\_value\_summary, [22](#), [33](#)

oneWayHoldout, [23](#)

prepare, [24](#), [25–27](#)

prepare.multinomial\_plan, [19](#), [20](#), [24](#), [24](#)

prepare.simple\_plan, [11](#), [24](#), [25](#)

prepare.treatmentplan, [6–10](#), [18](#), [21](#), [22](#),  
[24](#), [26](#), [29](#), [33](#)

print.multinomial\_plan, [28](#)

print.simple\_plan, [28](#)

print.treatmentplan, [29](#)

print.vtreatment, [29](#)

problemAppPlan, [30](#)

rmdatatable\_prepare, [30](#), [32](#)

rquery\_prepare, [3](#), [31](#), [31](#)

track\_values, [22](#), [25](#), [27](#), [32](#)

vnames, [33](#)

vorig, [34](#)

vtreat, [34](#)

vtreat-package (vtreat), [34](#)